Formal Methods for Reverse Engineering Gate-Level Netlists

Wenchao Li

Report Documentation Page			*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **18 DEC 2013**	2. REPORT TYPE	3. DATES COVERED **00-00-2013 to 00-00-2013**

4. TITLE AND SUBTITLE **Formal Methods for Reverse Engineering Gate-Level Netlists**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**48**	

Acknowledgement

The author would like to thank his collaborators Prof. Sanjit A. Seshia, Zach Wasson and Wei Yang Tan from UC Berkeley, Adria Gascon, Dr. Natarajan Shankar and Dr. Ashish Tiwari from SRI International, and Pramod Subramanyan and Prof. Sharad Malik from Princeton University. Some of the work done in this report could not have been possible without their assistance and guidance. The author would also like to thank Prof. Sanjit A. Seshia and Prof. Robert K. Brayton for reviewing this report and giving insightful feedback. The work described in this report was supported in part by Defense Advanced Research Projects Agency (DARPA) under the IRIS program, and by the Hellman Family Faculty Fund.

Formal Methods for Reverse Engineering Gate-Level Netlists

Wenchao Li

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Sanjit A. Seshia
Research Advisor

Date

* * * * * *

Professor Robert K. Brayton
Second Reader

Date

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Systems are increasingly being constructed from off-the-shelf components acquired through a globally distributed supply chain. There is a rising concern over the trustworthiness of these components, especially when used in mission-critical applications. The possibility that, a small but malicious modification could compromise the entire integrity, security and reliability of an integrated circuit (IC), is becoming a pressing concern. For example, the Integrity and Reliability of Integrated Circuits (IRIS) program from DARPA [3] seeks to develop techniques for deriving functions of digital, analog and mixed-signal ICs from limited operational specifications, as a way to ensure the overall integrity of a system possibly constructed using multiple third-party components. Such malicious modifications, commonly known as hardware trojans (HTs), can provide footholds for software based attacks, where the attacks are orchestrated by colluding software [18]. They can also provide side-channels for leaking sensitive information [23], or simply subvert the operation of the system under special conditions (e.g. special instruction sequences that trigger the trojan) [19].

In general, modifications can be introduced at different stages of the design and fabrication flow, and efforts specifically targeted at each stage have been made to mitigate these threats [36]. In this report, we consider the context where a logic alteration was made either at the Register Transfer Level (RTL) or at the gate level. At the RTL level especially, one can inject malicious behavior that can undermine the correct operation of the entire system with only a few lines of code written in a Hardware Description Language (HDL). Such design-time modifications are also difficiult to detect due to possible obfuscation [34] and small physical footprints [36]. Since they may be activated only under very specific conditions, they are unlikely to be triggered and detected in simulation or functional tests. Even if suspicion is raised during the operation or inspection of a system, currently there is no way to zoom into a particular portion (say the ALU unit) of a system by simply looking at the overall gate-level netilist. Most high-level structures such as word declaration, modularization, function separation are lost once the design has undergone logic synthesis, thus making it extremely difficult to perform targeted function search in the flattened netlist. In this report, we present *a systematic framework for automatically deriving high-level structures from the gate-level netlist of a digitial circuit.* First, we formally define the problem of reverse engineering a bit-level description into high-level description (henceforth referred to as REHLD) of a digital circuit. We then present several techniques for solving the REHLD problem.

Reverse engineering, if successful, can bring significant benefits. For example, identifying a *word-level* datapath allows the user to navigate the netlist at a higher level. Also, such word-level datapaths allow automatic graph-based inference techniques to be applied without complication from low-level details (e.g. the function that a particular gate implements). On the other hand, identifying the function (even partially) of a block of gates means one can now perform more comprehensive analyses to it in an isolated fashion. HT detection techniques, such as the ones presented in [18], are largely based on simulating the netlist. Therefore, such *dynamic* analysis techniques can be synergetically applied with the *static* techniques presented in this report, and in a more scalable manner. In addition to complementing existing HT and counterfeit detection techniques, reverse engineering can be applied directly to find IP violation, which is another rising concern in the semiconductor industry [14].

Algorithmic reverse engineering of an unstructured netlist, however, can be particularly challenging due to the following reasons.

- Lack of structure. A flattened netlist does not contain any of the hierarchy or module information that its RTL counterpart would typically have. In addition, optimization techniques such as multi-level logic minimization, technology mapping and retiming further destroy high-level structures in the netlist, and can result in overlapping functional blocks and gate sharing.

- Large functional space. Designs can vary from one to another and the space of possible functions that a circuit may implement quickly becomes intractable. This problem is further exacerbated by the assumption that only scanty documentation is given, which means well-defined and well-specified reference models are not available, thereby making it especially difficult to reason about the function of a netlist.

- Large implementation space. In case a high-level description of a function is given, e.g., the netlist contains an 8-bit multiplier, the space of all possible implementations for this particular function can be again enormous. The implication is that structural matching techniques quickly become impractical as the size of the target structure grows.

In this report, we present a systematic framework for automatically deriving high-level structures from the gate-level netlist of a digitial circuit, and techniques that specifically address each of these challenges. To cope with the large functional space, we crafted a library that contains more than a thousand commonly used components. Leveraging formal verification techniques such as symbolic evaluation, model checking and equivalence checking, we further address the problem of a large implementation space per function, and recover structure from an unstructured netlist.

1.2 Contributions

To summarize, we make the following contributions:

- We present a formal definition and a systematic framework for the netlist reverse engineering problem (REHLD), based on the notion of matching against *abstract components*.
- We describe a two-stage algorithm for deriving word-level datapaths in a bit-level netlist, which first finds candidate words and then propagates them using symbolic evaluation.
- We present an approach for matching an unknown sub-netlist against an abstract component library based on *mining behavioral patterns* from simulation or execution traces, followed by model checking.
- We present a second approach for matching an unknown sub-netlist against a known circuit using (conditional) equivalence checking – formulating the problem as solving a Quantified Boolean Formula (QBF) (as opposed to SAT in traditional equivalence checking). This formulation addresses the challenge of gate sharing in an optimized flattened netlist.
- We apply our approach to a collection of open-source designs. In particular, we demonstrate the effectiveness of our approach on netlists that contain up to 400,000 IBM 12SOI cells.

The rest of the report is organized as follows. We first describe the assumptions we make in this work in Section 1.3. Aterwards, we survey related work in Section 1.4, with a focus on algorithmic reverse engineering of netlists. Afterwards, we review background materials and formally define the REHLD problem. We then present techniques that try to address this problem from two angles – finding word-level datapaths in Section 2.3 and identifying functional modules in Section 2.4. Experimental results on applying these techniques to a number of circuit benchmarks are reported in Section 2.5. Finally, we conclude in Chapter 3.

The work described in this report is based on [22] and [21].

1.3 Assumptions

In this report, we work with the following assumptions for reverse engineeirng a gate-level netlist.

- RTL-level description of the original design is *not* available.
- Only limited operational specifications (e.g., a datasheet describing the high-level functionallities of a chip) are given.
- Micro-architectural and design-specific information is also *not* available.
- Information about how individual wires should be grouped together is only known at the primary input and output.
- The cell library for which the netlist is synthesized from is given.

1.4 Related Work

Digital circuit designers usually proceed from a high-level description to a gate-level netlist, and then to a physical layout and mask; it is rare to proceed in the opposite direction. However, as noted in Sec. 1.1, the study of reverse engineering of digital circuits has been gaining importance in recent years. We review some of the closely related work in this section. For a general survey on the taxonomy and detection techniques of hardware trojans, we point the readers to [36].

Hansen et al. [17] present a study of reverse engineering the well-known ISCAS-85 combinational circuits. They present several strategies, mostly manual, to reverse engineer circuit functionality from a gate-level schematic. Some of these include looking for common library components, repeated structures, computing truth tables of small blocks, and identifying bus structures and control signals. In this report, we provide a formal definition of the reverse engineering problem and present several automatic techniques for solving it. Particularly, our techniques reason with components that are at a much higher level of abstraction than those suggested by Hansen et al. Moreover, we demonstrate that they are effective in dealing with netlists that are several orders of magnitude larger than the ones they considered.

More recently, Subramanyan et al. [35] propose several techniques to identify high-level components such as register files, counters, adders and subtracters. Our work complements well with their solution. In fact, words generated by their bitslice aggregation algorithm are used as candidate words in our word propagation technique to infer more words. Our framework also provides additional features, such as the capability of navigating the netlist at the word level and that of handling gate sharing in module identification.

Our work does not address reverse engineering of arbitrary finite-state machine functionality. While any sequential circuit can be trivially viewed as a monolithic FSM, the challenge is to be able to decompose that FSM into a set of smaller FSMs, each of which performs a distinguishable function, thus making the resulting high-level description easier for a human to understand. The recent work by Shi et al. [33] is a step in this direction.

A key component of our work is to find the input-output signal correspondence between an abstract component and a block in the unknown circuit. There is not very much prior work in this area. Mohnke and Malik [26] present a BDD-based approach for comparing two combinational circuits whose input correspondence is not known apriori. The authors also extended their idea to finding latch correspondence for sequential circuits, by considering the combinational circuit computing the next-state function [25]; this does not address our problem, though, as sequential equivalence is required between the two circuits. Our approach, based on behavioral pattern matching followed by property checking, is a novel attempt in this direction.

Torrance and James [38] describe the practice of reverse engineering semiconductor-based products. Their approach includes product tear-downs (stripping packaging and disassembling the unit), "system-level analysis" (identifying components on a board and performing functional analysis through probing), process analysis, and circuit extraction (deriving a schematic from a stripped IC). Our work is complementary to this effort. Once a gate-level schematic is derived, our techniques can then be used to identify high-level components within the schematic.

Our technique addresses the REHLD problem for a system integrator who has *not* designed the circuit being reverse-engineered, but instead needs to verify its functionality prior to integration. We do not address the problem of untrusted manufacturing and IC piracy, where the designer is trusted, which can be tackled by techniques such as EPIC [31]. Our technique is complementary to other recent work on malicious trojan circuit detection (e.g., [18, 34]). We do not seek to find trojans, instead focusing on detecting if a sub-circuit exhibits correct behavior which is captured by a set of logical specifications; if our approach reports that a sub-circuit matches an abstract component, it is guaranteed to do so due to the use of formal verification. In addition, if the sub-circuit violates some security-related specification, our approach will report that as well.

Chapter 2

Reverse Engineering Gate-Level Netlists

2.1 Solution Overview

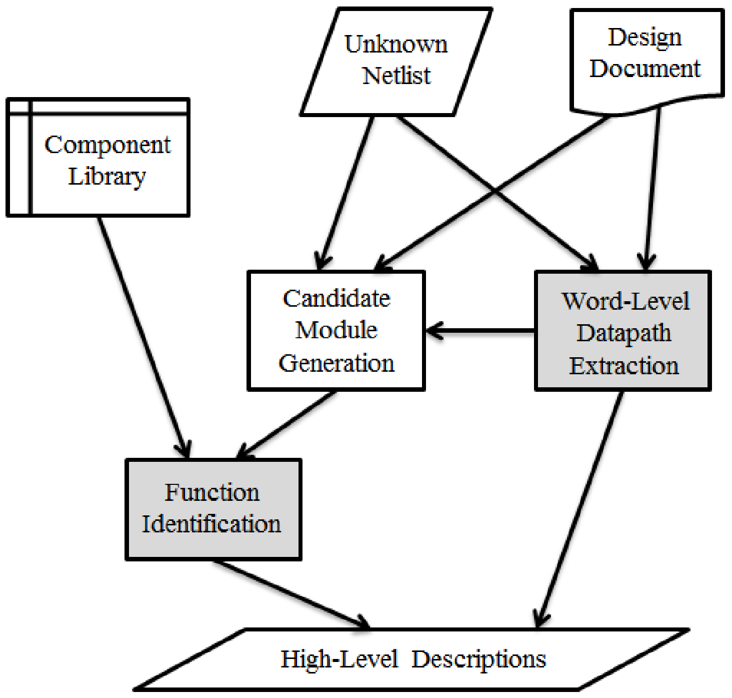


Figure 2.1: Netlist Reverse Engineering – Solution Overview

Figure 2.1 illustrates our proposed solution to the reverse engineering problem. Starting with an unknown netlist and a design document[1], we approach the problem from two

[1] We make no assumption on what additional information this document may provivde.

angles. First, we aim to uncover word-level dataflows from the bit-level netlist. Second, we try to associate individual parts of the netlist to components in a predefined library with known functionalities. The composition of these two is eventually produced as a high-level description of the netlist. In this report, we only partially address the problem of module identification – dividing the netlist into candidate modules where each of them will be mapped to a component in the library. We plan to pursue this direction further in future work.

2.1.1 Word-Level Datapath Extraction

A *word* is simply a bounded array of bits. A word-level dataflow is then a directed graph summarizing how one word is propagated to another word. Such word-level information is common at the RTL level. For example, below is a snippet of Verilog codes showing some simple word-level operations.

```
wire [7:0] a, b, c;
wire [3:0] d;
assign c = a & b;
assign d = {c[7:6],c[1:0]};
```

However, in a post-synthesis netlist, such information is lost and this is the first problem we will tackle in this report.

Figure 2.2 illustrates our two-stage approach to this problem.

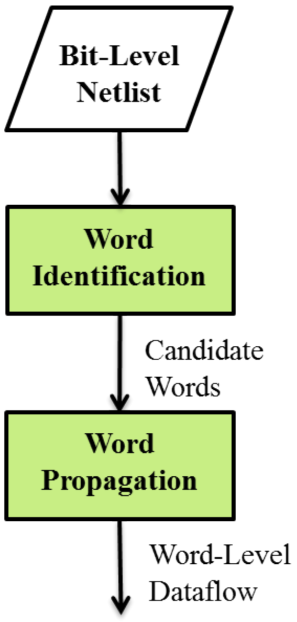


Figure 2.2: Overview of the Word-Level Dataflow Extraction

Given a bit-level netlist, the first stage identifies *candidate words*. We employ two techniques for solving this problem, one based on bitslice aggregation [35] and the other based on a notion called *shapehashing*. The first technique uses functional matching while the second one uses structural information to group "equivalent" wires into words. We discuss these techniques in detail in Section 2.3.1. Starting with the candidate words and other known words (such as ones at the primary input and output), the second stage infers more words by iteratively propagating them across gates in the netlist, by leveraging ideas in symbolic evaluation. We describe this in detail in Section 2.3.2. These words can also be used as boundaries to form a netlist *slice*, which naturally provides a divide-and-conquer strategy to reason about separate parts of the overall netlist.

2.1.2 Function Identification

The second collection of techniques we present in this report is known as function identification. Essentially, given a slice of the netlist, the task is to identify the functionality that this slice implements. We reduce this problem to a verification problem – checking whether the netlist slice is functionally equivalent to a known component. To deal with the large space of possible functions, we created a database consisting of more than a thousand circuit components inspired by common design patterns, ranging from simple arithmetic circuits such as adders to communication protocols such as I^2C. This component library by no means captures all possible functionality a netlist may contain. However, it serves as a testbed for us to experiment with and evaluate the proposed approach. We give more detail about this library in Section 2.4.1.

The key steps of the overall approach are illustrated in Figure 2.3. Given this component library, we pursue two different ways to functionally identify if a netlist slice matches a component in this library.

The first is based on property checking – model checking if the netlist slice satisfies a set of logical properties associated with the component. The added challenge here, compared with similar verification task, is that the correspondence of inputs and outputs between the two circuits are not known. Our solution is to use a pattern mining technique to heuristically determine this correspondence. The general function of the unknown netlist slice is then determined by finding the closest match in the component library, by model checking the unknown netlist slice against each logical specification.

The second approach is based on equivalence checking. It is well known that the equivalence checking problem (for combinational circuits) can be formulated as a SAT problem [16]. However, in the context of reverse engineering, gate-sharing (i.e. overlapping functional blocks) is quite common. Specifically, this results in additional *side inputs* to the (smallest) netlist slice that implements a particular function.

To address this problem, we extend equivalence checking to handle side wires, by formulating the problem as a Quantified Boolean Formula (QBF). We describe this formulation in more detail in Section 2.4.3.

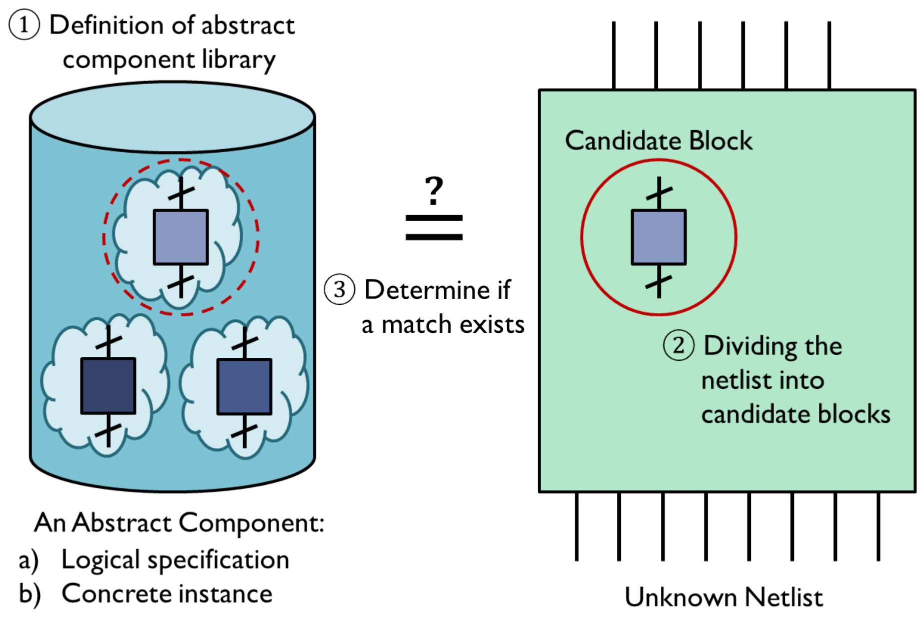

① Definition of abstract component library
? ≡
③ Determine if a match exists
Candidate Block
② Dividing the netlist into candidate blocks
An Abstract Component:
a) Logical specification
b) Concrete instance
Unknown Netlist

Figure 2.3: Overview of Function Identification

2.1.3 Example – CMP Router

To illustrate the overall approach, consider the following example.

Example 1. *Consider the high-level description of a chip multiprocessor (CMP) router as shown in Figure 2.13. It is a composition of four high-level modules. The* input controller *comprises a set of FIFOs buffering incoming* flits *and interacting with the* arbiter*. When the arbiter grants access to a particular output port, a signal is sent to the input controller to release the flits from the buffers, and at the same time, an allocation signal is sent to the* encoder *which in turn configures the* crossbar *to route the flits to the appropriate output ports. Our goal is to reverse engineer such high-level information including the flow of flits and function of FIFOs, as captured in Figure 2.13, from its gate-level netlist.*

2.2 Problem Definition

We begin with some basic definitions and terminologies in Section 2.2.1, followed by the formal problem definition in Section 2.2.2.

2.2.1 Preliminaries

Circuits and Netlists

In this report, we consider the following two abstractions of modeling digital designs.

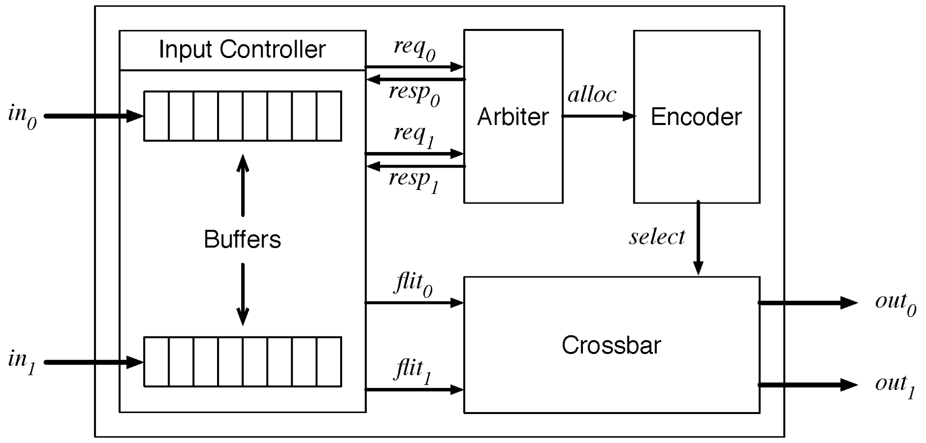


Figure 2.4: CMP Router comprising four high-level modules

A *sequential circuit* $\mathcal{C}$ is a tuple ($\mathcal{I}$, $\mathcal{O}$, Q, Q_0, δ, θ) where

- $\mathcal{I}$ is a finite set of input values;
- $\mathcal{O}$ is a finite set of output values;
- Q is a finite set of states;
- $Q_0 \subseteq Q$ is a finite set of initial states;
- $\delta : Q \times \mathcal{I} \to Q$ is the transition function, and
- $\theta : Q \to \mathcal{O}$ is the output function.

A *gate-level netlist* $\mathcal{N}$ is a set of registers R (state-holding elements) and combinational gates G interconnected by wires W. Each gate $g \in G$ is associated with a Boolean function f_g with input pins I_g and output pins O_g. Such a netlist $\mathcal{N}$ with I input wires and O output wires is naturally associated with a circuit $\mathcal{C}_{\mathcal{N}}$ as follows. The input space $\mathcal{I} = 2^I$, the output space $\mathcal{O} = 2^O$, and the state space $Q = 2^R$. The transition and output functions are defined correspondingly by the logic of the gates.

A collection of registers and gates naturally induces a *netlist slice* (denoted as $\mathfrak{n}$, i.e. a portion of the original netlist. Let $\mathcal{C}_{\mathfrak{n}}$ be the circuit associated with $\mathfrak{n}$. In this report, we consider slices that do not contain disjoint slices. For simplicity, we henceforth use circuit(s) for sequential circuit(s) and netlist(s) for gate-level netlist(s).

An *input-output trace* (or simply, a *trace*) τ of a circuit is a sequence of input-output values $(i_0, o_0)(i_1, o_1)\ldots$[2] starting from a state q_0 such that $o_j = \theta(q_j)$ and $q_{j+1} = \delta(q_j, i_j)$ for all $j \geq 0$. A *finite trace* τ^k of length k is then a sequence of values $(i_0, o_0)(i_1, o_1)\ldots(i_{k-1}, o_{k-1})$.

[2] For simplicity, we assume there is a single clock in the digital circuit and restrict ourselves to values recorded at the rising edge of the clock.

Word-Level Datapath

Words are desirable in datapath representations, especially for ease of human understanding. In a RTL description of a circuit, many signals are moved together and operated on at the word-level. However, identifying such word-level structures in a netlist poses significant challenges, since all the signals are given at the bit-level.

We use the vector notation $\vec{w} = [w_0, \ldots, w_{k-1}]$ to denote an ordered set of (bit-level) wires with cardinality k. For convenience, we use $\vec{w}_\eta$ to denote the word formed by taking the individual bits of $\vec{w}$ according to and as ordered by an index array η. For example, if $\eta = [3, 0]$, then $\vec{w}_\eta = [w_3, w_0]$.

It is also convenient to view word-level datapaths as a directed graph $\mathcal{G}$, where the vertices represent words, and an edge $(\vec{u}, \vec{v})$ represents the word $\vec{u}$ propagating to $\vec{v}$ (across a single layer of gates).

Formal Specification

A *formal specification* of a sequential circuit $\mathcal{N}$ is a set $\mathcal{S}$ of input-output traces of that circuit. Intuitively, every trace in $\mathcal{S}$ is an allowed behavior of $\mathcal{N}$ and every trace outside $\mathcal{S}$ is disallowed.

There are broadly two ways to write a formal specification for a digital circuit. The automata-theoretic approach is to describe $\mathcal{S}$ as a finite-state machine over infinite input sequences [37]. The other approach is to write a logical formula (or set of formulas) characterizing the input-output behavior of the circuit. The latter approach has gained favor in the EDA community over the years, especially in the form of assertion languages that allow one to specify temporal properties of a system, and which are usually slight extensions of *linear temporal logic* (LTL) [29]. We follow this logic-based approach in this report. Abusing notations, we also use $\mathcal{S}$ to denote the LTL formula characterizing the input-output behavior of a circuit.

A brief overview of LTL is provided below. A LTL formula is built from atomic propositions AP, Boolean connectives (i.e. negations, conjunctions and disjunctions), and temporal operators $\mathbf{X}$ (*next*) and $\mathbf{U}$ (*until*). Given an atomic proposition $p \in AP$, a formula in linear temporal logic (LTL) can be constructed as follows.

$$\psi ::= p \,|\, \neg\psi \,|\, \psi \vee \psi \,|\, \mathbf{X}\ \psi \,|\, \psi_1 \mathbf{U}\ \psi_2$$

Other temporal operators $\mathbf{F}$ (*eventually*) and $\mathbf{G}$ (*globally*) can be derived using the temporal operators $\mathbf{X}$ and $\mathbf{U}$, and Boolean connectives: $\mathbf{F}\ \psi = \texttt{true}\mathbf{U}\ \psi$ and $\mathbf{G}\ \psi = \neg\mathbf{F}\ \neg\psi$.

For example, we can express that "every request must be eventually followed by a grant" in LTL as $\mathbf{G}$ (request $\Rightarrow$ $\mathbf{F}$ grant), where the operator $\mathbf{G}$ specifies that globally at every point in time a certain property holds, and $\mathbf{F}$ specifies that a property holds either currently or at some point in the future.

Given a netlist $\mathcal{N}$ and a LTL specification ψ, we can *verify* if $\mathcal{C}_\mathcal{N} \models \psi$. This is commonly known as LTL model checking. In this report, we assume the readers are familiar with the notion of property checking and have knowledge of LTL model checking. More details about model checking and LTL model checking can be found in [11] and [32].

Abstract Components

An *abstract component* α is a triple $(I, O, \mathcal{S})$, where I and O are sets of input and output signals, respectively, and $\mathcal{S}$ is a formal specification defining allowed input-output behavior of the component. An *instance* of an abstract component α is any circuit or netlist that satisfies the specification $\mathcal{S}$ of α.

We illustrate the notion of an abstract component using an example.

Example 2. *Consider an arbiter servicing two (input) request lines r_0 and r_1 with two grant lines g_0, g_1 as outputs. The abstract arbiter component would comprise the following LTL properties:*

$$\boldsymbol{G}\,\neg(g_1 \wedge g_2)$$
$$[\boldsymbol{G}\,\boldsymbol{F}\,\neg(r_0 \wedge r_1)] \Rightarrow \boldsymbol{G}\ (r_0 \Rightarrow \boldsymbol{F}\,g_0)$$
$$[\boldsymbol{G}\,\boldsymbol{F}\,\neg(r_0 \wedge r_1)] \Rightarrow \boldsymbol{G}\ (r_1 \Rightarrow \boldsymbol{F}\,g_1)$$

The first property states that both requests cannot be granted at the same cycle. The last two properties state that a request must eventually be granted, provided there are infinitely many cycles where no competing requests are present – the latter assumption is the property $\boldsymbol{G}\,\boldsymbol{F}\,\neg(r_0 \wedge r_1)$.

High-Level Description

Informally, a high-level description of a netlist is a composition of *abstract components*. This includes a mapping from netlist slices to abstract components, as well as the datapaths that connect these components.

A *library* of abstract components (*component library*, for short) $\mathcal{L}$ is a set $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ of abstract components. We will assume that each abstract component is also accompanied by at least one concrete instance; this is reasonable, as the abstract component library is typically constructed from observations of commonly occurring components in hardware designs.

Example 3. *Examples of abstract components include common hardware design patterns and modules such as an arbiter, FIFO buffer, adder, multiplier, content-addressable memory (CAM), and crossbar. Abstract descriptions of finite-state machines are relevant for circuits that implement protocols; e.g., transmitter or receiver modules implementing the Ethernet or I^2C protocols.*

A *high-level description* (HLD) of a netlist $\mathcal{N}$ with input signals I and output signals O is a tuple (I, O, Γ) where Γ is a set of instances of abstract components drawn from $\mathcal{L}$ plus some "glue" logic such that the *synchronous composition* of these instances and the "glue" logic is (sequentially) *equivalent*[3] to the original netlist $\mathcal{N}$.

2.2.2 Problem Definition

We are now ready to formally define the problem of reverse engineering a high-level description (REHLD) of a circuit.

Definition 2.1. *Given*

- *an* unknown *netlist* $\mathcal{N}$ *containing cells* R *and* G *interconnected by wires* W*,*
- *a library* $\mathcal{L} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ *of abstract components,*

the REHLD problem for $\mathcal{N}$ *is to derive a high-level description* $\mathcal{N}' = (I, O, \Gamma)$ *where* $\mathcal{N}'$ *is equivalent to* $\mathcal{N}$ *and* Γ *is a composition of instances of abstract components from* $\mathcal{L}$ *and a collection of registers* $R' \subseteq R$*, gates* $G' \subseteq G$ *and wires* $W' \subseteq W$ *that make up the "glue" logic.*

Solving the REHLD problem involves multiple steps. We rephrase each of these steps using the notation introduced in this section. There are four main steps:

- *Word-level datapaths extraction:* Extract word-level datapaths $\mathcal{G}$ from the bit-level netlist $\mathcal{N}$.
- *Library definition:* Constructing an abstract component library $\mathcal{L}$;
- *Netlist slice identification:* Extract from the given unknown netlist $\mathcal{N}$ a set of netlist slices $n_1, n_2, \ldots, n_k$, where each n_i can be used as a candidate for matching against the component library $\mathcal{L}$;
- *Matching against component library:* Given a candidate netlist slice n and an abstract component library $\mathcal{L} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, determine whether there exists an abstract component α_i such that:
  (a) *Input-output correspondence:* Each input (respectively, output) signal of α_i that appears in the formal specification of α_i is mapped 1-1 to some input (respectively, output) signal of n.
  (b) *Verification:* n is an instance of α_i; i.e., $\mathcal{C}_n$ satisfies the specification $\mathcal{S}_i$ associated with α_i.

These steps can be categorized into datapath extraction (Step 1) and function identification (Step 2, 3 and 4). In the rest of this report, we first describe how word-level datapaths can be extracted from a bit-level netlist. Afterwards, we present two novel approaches for finding the mapping from netlist slices to abstract components.

[3]Equivalence is defined with respect to the semantics of the associated circuit $\mathcal{C}_{\mathcal{N}}$ of $\mathcal{N}$.

2.3 Finding Word-Level Datapath

In this section, we describe a systematic approach for identifying word-level datapaths in a (bit-level) netlist. The approach involves a combination of two sub-procedures – *word identification* and *word propagation*. In word identification, the goal is to find *candidate* groupings of wires that may form words. Starting with the candidate words and other known words (such as ones at the primary inputs and outputs), the second procedure tries to infer more words by iteratively propagating them across gates in the netlist.

2.3.1 Word Identification

We employ two techniques for finding candidate words in a netlist. The first is based on bitslice aggregation [35], and the second is based on a notion called *shapehashing*. On a high level, the first technique uses functional matching while the second uses structural information to group "equivalent" wires into words.

Bitslice Aggregation

This technique is not our contribution and is described in detail in [35]. We briefly review it in this section.

The function of a wire w in the netlist can be characterized by a *feasible cut* of w. This is defined as a set of wires in the transitive fan-in cone of w such that an arbitrary assignment of truth values to each wire in the set completely determines the value of w [10]. A cut is said to be *k-feasible* if it has no more than k inputs. As in [35], we enumerate the set of 6-*feasible cuts* for each node. Each cut then forms a *bitslice* rooted at w, which is a Booolean function with a single output and no more than 6 inputs.[4] Once all the bitslices are identified, they can be grouped into equivalence classes using permutation-independent Boolean matching. For example, a bitslice matching the function $y = a \wedge b \vee c$ and a bitslice matching the function $y = b \wedge c \vee a$ are grouped into the same class.

Now that we have found all the duplicated bitslices across the netlist that compute a particular function, we can look for *aggregates* of them that are connected in specific patterns. Following the approach described in [35], we group all matching bitslices that (1) have a common select signal; (2) the output of one bitslice feeds to the input of another (e.g. carry chain in a ripple carry adder). Since aggregated bitslices are essentially circuits that operate on sets of nodes simultaneously. We group the inputs or outputs of aggregated bitslices together to form candidate words. Note that in the case of a carry chain, the words are *ordered* in the carry direction.

[4]The number 6 is chosen for efficiency reasons, as the number of cuts for $k > 6$ becomes significantly larger in our experience. Also, most common bitslices have less than six inputs, e.g., a full adder bitslice has 3 inputs [35].

Shape Hashing

The idea of *shape hashing* is to assign each wire in the netlist a *shape*, and then create a hash function for all the shapes such that we can easily identify equivalent wires if they have the same shape. The *shape* of a wire w is defined as the directed graph formed by the set of gates *backward reachable*[5] from w. A *k-bounded shape* is simply a shape where all the gates in the graph are backward reachable from the root w within k steps, and at least one of them is reachable at exactly k steps. When the directed graph is cyclic, we unroll the loops by duplicating gates along the loops until the gates are not backward reachable from w in k steps. In our experiments, we used all values of $k \in \{2, 3, 4\}$[6].

To compute a hash key from each shape efficiently, we perform a depth-first-search traversal of the DAG (i.e. shape) backward starting from w to produce a serialization of the DAG using gate and wire types. Multiple children gates in the traversal are tie-broken lexicographically. However, we do not check for graph isomorphism, especially one that is induced by having gates with commutative ports, for efficiency reasons.

We further refine the equivalence classes by taking into account the relative location of the wires. We define the distance between two wires as the smallest number of gates between them in the netlist[7]. With this distance measure, we form equivalence subclasses for wires that have the same shape. In each subclass, the wires are located to one another by at most d distance (number of gates). The collection of wires in each subclass then forms a *candidate word*. Individual wires of the same word are ordered arbitrarily.

2.3.2 Word Propagation

In this section, we describe the second important piece of the overall word-level datapath extraction process – an algorithm for propagating words. Intuitively, we want to see if arbitrary values of a word can propagate across a set of gates to reach a new set of wires. To do this efficiently, we use symbolic evaluation [9], which allows the evaluation of a set of values simultaneously in a single run.

Similar to Roth's D-calculus [30], we redefine the functions of the logic gates in the netlist to operate over the expanded domain $\{0, 1, D, \overline{D}, X\}$, where D represents a symbolic value in $\{0, 1\}$, $\overline{D}$ is the negation of D, and X represents an unknown/undetermined value. Figure 2.5 shows two examples of symbolic evaluation for basic Boolean gates.

Our solution for word propagation uses a "guess-and-check" approach. Starting from some known or candidate word (source word), we first try to find a set of wires (target word) whose cardinality is no greater than the cardinality of the source word, that are located one level of gates away from this word. For simplicity of discussion, we search *forward* for such a

[5]In general, one can use both forward and backward reachable gates to assess structural similarity. In our case, the set of backward reachable gates also describes a Boolean function with a single output at w.

[6]As k increases further, even wires originally declared as parts of the same word become structurally dissimilar in an optimized netlist.

[7] A gate can be traversed to either from an input or from an output of the gate.

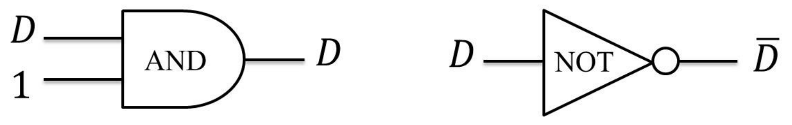
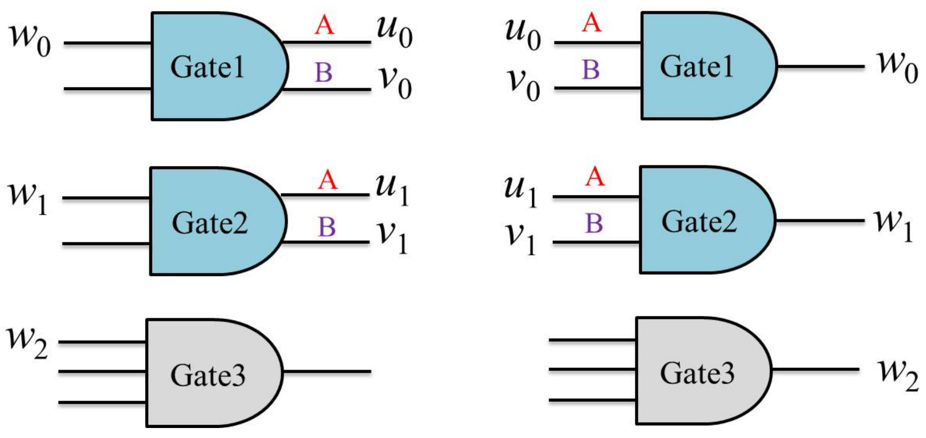


Figure 2.5: D-calculus Examples for Boolean “AND” and “NOT”



(a) Forward Search (b) Backward Search

Figure 2.6: Structural Heuristics for Finding Target Words from Source Word $\vec{w}$. Gates of different function types are colored differently.

set of wires, using a procedure called *FindForwardWordPairs*. This set of wires constitutes a target word, i.e. the word to propagate to. Next, we construct a netlist slice for symbolic evaluation using *SetupSymbolicEvaluation*. Finally, we check if symbolic values of the source word can indeed be propagated to the target word, by using the procedure *TryPropagate*.

FindForwardWordPairs, i.e., the “guessing” stage of the algorithm, consists of finding promising target words. Figure 2.6 shows two structural heuristics we use to find target words forward and backward. The idea is to group gates according to their function types and group wires according to the ports they connect to. For instance, consider the source word $\vec{w}$ as shown in Figure 2.6a, which is 3-bit wide, and assume that “Gate1” and “Gate2” have the same function type, we guess two target words $\vec{u}$ and $\vec{v}$, each of 2-bit wide. We then check if the subword $\vec{w}_{[0,1]}$ can propagate to $\vec{u}$ and $\vec{v}$ respectively, using Roth’s D-calculus.

SetupSymbolicEvaluation is the first step of the “checking” part of our algorithm. In general, if we just evaluate the gates between the source word and the target word by assigning X to the other inputs of these gates, we would not be able to propagate to the target word. The key insight here, which leads to effective word propagation as we will show in Section 2.5.2, is that if the target word is indeed a word (in the sense that it would be naturally declared as a bit-vector in a RTL description language like Verilog), then often there exist a few *nearby* (in the fan-in cone of these gates) wires which behave as *control* *signals*. This means that for some specific concrete assignments to these control signals, the target word will take the values of the source word (or their negation). This pattern is quite common in digital circuits, such as in the case of a multiplexer, or a conditional assignment in an “if-then-else” statement. In fact, even if the condition involves many signals and Boolean operators, the synthesis tool will likely create a wire in the netlist that is the evaluation result of this condition. We elaborate on how we make use of this insight in Figure 2.7.

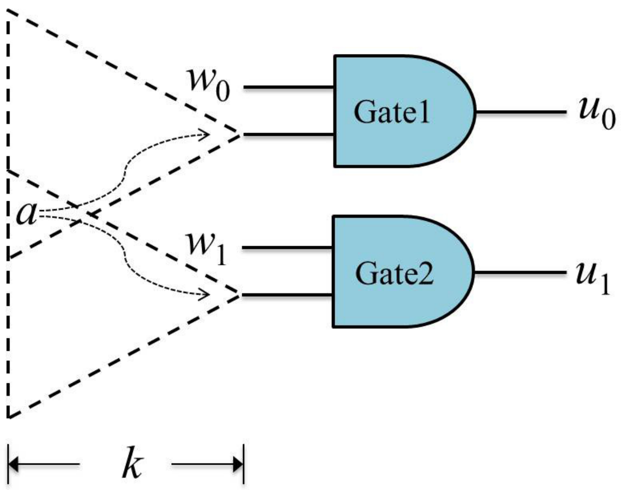


Figure 2.7: To check if word $w = [w_0, w_1]$ propagates to word $u = [u_0, u_1]$, we find potential control signals such as a and include them as inputs in the netlist slice on which symbolic evaluation is performed.

We consider wires and gates in the fan-ins of “Gate1” and “Gate2”, up to some small depth k. Any wire that lies in the intersection of these fan-ins is treated as *control*, e.g. wire a. We construct a netlist slice including the gates between the source word and the target word, as well as the gates in the aforementioned fan-ins. The fan-in gates are aggregated in a backward traversal manner up to depth k or when a sequential logic cell such as a flip-flop is reached. This ensures that the overall netlist slice forms a combinational circuit. This is the netlist that we will symbolically evaluate, and is set up in the following way.

- Source word: each bit is assigned the symbolic value D.
- Control wires: If the number of control wires is greater than 3, we enumerate all combinations of wires of size 3 in the set of control wires. In each combination, we further try all possible concrete assignments (assignments using 0s and 1s only) to the wires involved. The choice of the constant value 3 is related to the insight mentioned above that the synthesis tool will likely add wires capturing the evaluation of complicated conditions.
- Other inputs: each wire is assigned the unknown value X.

TryPropagate is the second piece of the "checking" stage of the algorithm. For each set of concrete assignments to condition wires, the netlist slice is symbolically evaluated afresh using the D-calculus. Because the netlist slice is a combinational circuit, symbolic evaluation can be done efficiently by evaluating each gate in the slice in a topological order. If every bit of the target word is evaluated to D or $\overline{D}$ for any concrete assignment to the control wires, then the target word is considered as a *true* word. In this case, the propagation process iterates and tries to use this new word to infer more words.

The overall algorithm is summarized in Algorithm 1. For simplicity, we show the only parts for forward propagation. The algorithm iterates over the set of source words W. At each iteration, a word $\vec{w}$ is removed from the source pool. *FindForwardWordPairs* then uses the structural heuristics described in Figure 2.6a to find promising pairs of words to test for propagation. For each pair of source word $\vec{u}$ and target word $\vec{v}$, if no propagation has been attempted so far from $\vec{u}$ to $\vec{v}$, the pair is added to $\mathcal{H}$ and we proceed to checking if $\vec{u}$ can propagate to $\vec{v}$. This is achieved by *SetupSymbolicEval* which first sets up the netlist slice C' and control wires cw for symbolic evaluation as described in Figure 2.7, and *TryPropagate* which then evaluates C'. If propagation succeeds, this means we have verified $\vec{u}$ and $\vec{v}$ are words and we add them to the set of inferred words $\mathcal{W}'$. Since $\vec{v}$ may be further propagated forward, we add it to the set of source words $\mathcal{W}$. The algorithm terminates when we have tried all words that can be inferred from, i.e. $\mathcal{W}$ is depleted.

Algorithm 1 Symbolic Evaluation for Propagating Words Forward

```
1: **Input:** the set of candidate/source words $\mathcal{W}$, netlist $C$.
2: **Output:** the set of inferred words $\mathcal{W}'$.
3: **Initialize:** set $\mathcal{H}$ to $\emptyset$.
4: **while** $\mathcal{W} \neq \emptyset$ **do**
5:     $\vec{w} = pop(\mathcal{W})$
6:     $\mathcal{P} = FindForwardWordPairs(\vec{w})$
7:     **for** $(\vec{u}, \vec{v}) \in \mathcal{P}$ **do**
8:         **if** $(u, v) \notin \mathcal{H}$ **then**
9:             Add $(\vec{u}, \vec{v})$ to $\mathcal{H}$.
10:            $(C', cw) = SetupSymbolicEval(C, \vec{u}, \vec{v})$
11:            **if** $TryPropagate(C', cw, \vec{u}, \vec{v})$ succeeds **then**
12:                Add $\vec{u}, \vec{v}$ to $\mathcal{W}'$.
13:                **if** $\vec{v} \notin \mathcal{W}$ **then**
14:                    Add $\vec{v}$ to $\mathcal{W}$.
15:                **end if**
16:            **end if**
17:        **end if**
18:    **end for**
19: **end while**
```

2.3.3 Discussion

In this section, we remark on the potential limitation and robustness of the proposed techniques, and discuss open problems.

- Order of bits in a word. In general, it is difficult to infer the ordering of bits in a word as it was originally described (e.g, in RTL). Other than a few special cases, such as in a carry chain or at the primary I/O, an arbitrary order is picked for a candidate word during word propagation. Due to the generally large number of candidate words produced, we hash any new word produced in the propagation step *unordered*. This prunes out a significant portion of the search space since the search stops if the same collection of bits is encountered. However, the tradeoff is that we may miss re-ordering operations on words. This also has implications on function identification when a candidate module is generated using propagated words as boundaries. For example, if we want to match a candidate module to an adder, we need to know the *correspondence* of the input and output bits of the module to those of the adder. We describe a technique that addresses this problem in Section 2.4.2.

- Robustness of structural heuristics. Even though we start with an *unstructured* netlist, we use structural heuristics to aid both word identification and propagation – grouping wires based on *shapes* and grouping gates based on types of gates. This is based on our observation that certain regularities still remain after the circuit is synthesized and optimized. For example, the same type of gates (and respectively the same input/output pins of these gates) are often used when a word is propagated to another word. In fact, this grouping heuristics allows us to identify instances when only part of a word is propagated or when a word is propagated across one level of gates to two different words. In our experience, the simple structural heuristic we use in Section 2.3.2 does not result in much reduction in performance (number of words found). Alternatively, we can expand the domain further by assigning each wire a different symbolic value, and then check if the same value is obtained at an output of the gate(s) that the wire leads to. However, in case several target wires take the same symbolic value, a similar grouping based on gate and pin types would have to be performed to distinguish these wires.

  In the context of dealing with (maliciously) modified circuits, we assume that the modification is introduced before logic synthesis (e.g., in the Verilog RTL description). This means that the structure of the post-synthesis is not completely obfuscated, thus allowing us to exploit structural similarities to heuristically guide our techniques. While this assumption does not apply to all attacker models, it is largely valid from an economic perspective. Changing the function of a post-synthesis netlist also requires the attacker to satisfy other design constraints, such as timing and capacitive balancing. Hence, while it is theorectically feasible to apply arbitrary modifications to a circuit at the netlist (or even mask) level, the attacker may not have enough incentives to do so.

- Reachability of propagating conditions. During word propagation, we check if a word can be symbolically propagated to another word under some assignment to a set of condition wires. Theorectically, one needs to check if these values are reachable. However, this would incur significant overhead in runtime since a large number of these reachability tests are needed and we are dealing with netlists with thousands to millions gates and registers. As a compromise, we use a small number of condition wires and propagate words optimistically. This compromise is also motivated by the observation that a conditional assignment in the RTL level typically translates to a single or a few wires (even if the condition is a large Boolean expression) taking the values of the condition in the netlist.

2.4 Functional Module Identification

In this section, we first describe our library of abstract components, and then present two techniques, both based on reduction to formal verification problems, for finding a match of a given netlist slice against the component library.

2.4.1 Library of Abstract Components

In general, a burden is placed on the end user to create an initial component library. While this is a limitation for any library-based approach, common hardware design patterns [12] and standardized interfaces are ideal starting components for the library. Our component library currently contains the following types of circuits (the same type of circuits may be parameterized by, for example, the width of inputs and outputs), ranging from simple multiplexer to complex controllers.

- Multiplexer and demultiplexer
- Boolean operations, including NOT, AND, OR and XOR
- Comparator
- Shifter
- Modulo operation
- Adder and subtractor
- Multiplier and divider
- Counter
- Decoder and encoder
- Register file

- FIFO
- Arbiter
- Arithmetic core
- Communication interfaces/controller
- Memory controller
- DSP core
- Error correcting code
- Audio decoder
- GPIO
- Video controller
- Processor core

For ease of access and future exploration, each component is associated with the following information.

- Number of input bits;
- Number of output bits;
- Port declaration in words, e.g. input in1 [7:0];
- Behavioral (RTL) Verilog description;
- Synthesized Verilog netlist using the IBM 12SOI cell library;
- A Verilog test bench;
- Formal specificaton $\mathcal{S}$ of the component.[8]

Currently, we have gathered a total of 1285 different circuits and stored them in a MySQL database.
Remark. The simpler components are generally more useful due to the relatively smaller sizes and smaller variations in implementations. A significant effort is also required if one wishes to fully specify a circuit using formal specifications. However, we believe partial specification is still useful, since knowing what properties a netlist slice satisfies (and does not satisfy) is itself a stride towards understanding the high-level function of the netlist.

[8]We have only created formal specifications for the benchmarks used in the experimental section of this report.

2.4.2 Matching by Property Checking

In this section, we first show how temporal patterns mined from simulation or execution traces of the circuits can be used to determine input-output correspondence. Afterwards, we use *model checking* [11] to determine whether a given netlist slice satisfies the formal specification associated with an abstract component. The approach is illustrated in Figure 2.8.

Input-Output Correspondence

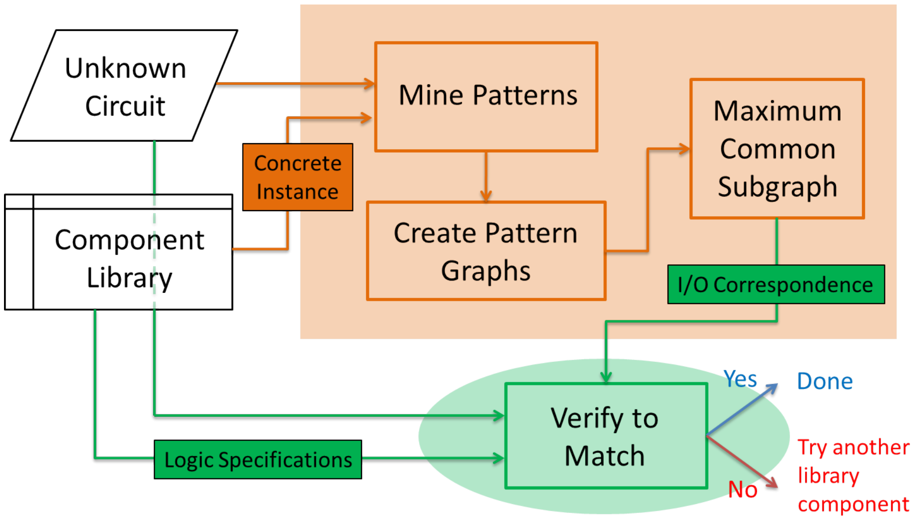


Figure 2.8: Overview of Component Matching by Property Checking

Given an unknown sub-circuit $\mathfrak{n}$ and an abstract library component α, we must first compute the correspondence between their input and output signals, if one exists. A brute-force approach will have to try all possible permutations of the signals. In addition, the numbers of inputs and outputs of the two circuits may not be identical. We use a heuristic procedure for this step. First, a concrete instance of α (e.g. a reference circuit that satisfies that formal specification of α), denoted as $\mathfrak{n}'$ is obtained. Then, given the two circuits $\mathfrak{n}$ with interface signals $\mathcal{V}_{\mathcal{I}} = I \cup O$ and $\mathfrak{n}'$ with interface signals $\mathcal{V}_{\mathcal{I}}{}' = I' \cup O'$, our method tries to find the corresponding signals between $\mathcal{V}_{\mathcal{I}}$ and $\mathcal{V}_{\mathcal{I}}{}'$.

Definition 2.2. *Given two sets of signals A and B, the signal correspondence between A and B is a bijective mapping $\sigma : \bar{A} \to \bar{B}$, where $\bar{A} \subseteq A$ and $\bar{B} \subseteq B$. We say the mapping is* maximum *if there does not exist another mapping $\sigma' : \tilde{A} \to \tilde{B}$ such that $\bar{A} \subset \tilde{A} \subseteq A$ or $\bar{B} \subset \tilde{B} \subseteq B$.*

Our approach to finding a good signal correspondence (so that the inputs and outputs of two equivalent circuits are matched correctly) on mining patterns from a set of input-output traces of the two circuits. Other approaches, e.g., based on the structure of the circuits, are also possible, and will be left to future work.

The key insight in our approach is that two signals are *similar* if they exhibit similar behaviors in relation to other signals. In this paper, we measure the similarity of two signals by checking if they satisfy some particular patterns (in relation to other signals) in the traces. However, our framework is general – one can use other definitions of similarity such as statistical measures.

In a nutshell, our method uses a two-step combination of *pattern mining* and *graph matching*. The pattern mining step infers *likely* temporal properties of a circuit from input-output traces of that circuit. It is important to note here that for the unknown component, we can only observe the trace induced by a test bench at the chip-level. This means that it is not possible to control the simulation as in the case of the known component. Consequently, even if the unknown circuit is identical to the library component, the behaviors of the two traces can be very different. In this work, we use pattern mining as a way to concisely capture key features of a trace. The mined properties are represented in terms of a *pattern graph*. Given circuits $\mathfrak{n}$ and $\mathfrak{n}'$, we compute pattern graphs for each of them. Then, a graph matching procedure is used to compute the *maximum common subgraph* between these two graphs, which yields the desired maximum bijection. We elaborate below.

Pattern Mining

We follow the approach described in [20] for mining temporal patterns from input-output traces of a circuit. A pattern is defined over (delta) events, and is given either as a LTL formula or a regular expression. An *event* e is a tuple $\langle \vec{s}, \vec{v}, t \rangle$, where $\vec{s}$ is a set of signals and $\vec{v}$ is the corresponding valuations at cycle t. We denote the valuation of a Boolean signal s at cycle t as $v_{s,t}$. A *delta event*, denoted Δe, is an event such that at least one of its constituent signals changes value from the previous valuation, i.e. $\Delta e := \langle \vec{s}, \vec{v}, t \rangle$ such that $\exists\ s \in \vec{s}, v_{s,t} \neq v_{s,t-1}$. In this report, we use the follow simple pattern templates.

- **Alternating (A)** An alternating pattern between two delta events Δa and Δb is true when each occurrence of Δa alternates with an occurrence of Δb. Note that this does not mean Δb follows Δa immediately in the next cycle. This pattern can be described by the regular expression $(\Delta a\, \Delta b)^*$.

- **Next (X)** The next pattern corresponds to the LTL formula "$\mathbf{G}\ (\Delta a \Rightarrow \mathbf{X}\ \Delta b)$." One can easily generalize this pattern to fixed-delay pairs.

- **Until (U)** The until pattern can be used to describe behaviors such as "the request line stays high until a response is received." Figure 2.9 shows a trace where this pattern is satisfied. Formally, the LTL formula is "$\mathbf{G}\ (a_{0\to1} \Rightarrow \mathbf{X}\ (a\ \mathbf{U}\ b_{0\to1}))$."

- **Eventual (F)** The eventual pattern can be described by the LTL formula "$\mathbf{G}\ (\Delta a \Rightarrow \mathbf{X}\ \mathbf{F}\ \Delta b)$."

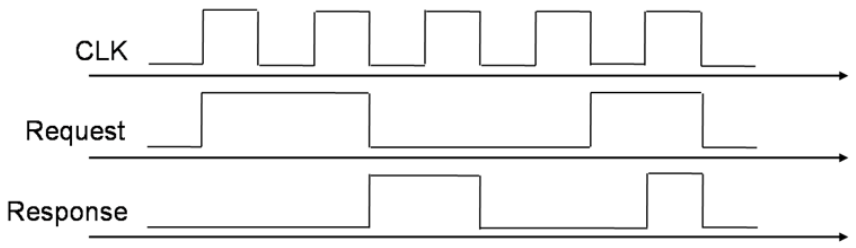


Figure 2.9: Request stays high until a response is received.

The idea of pattern mining is similar to online monitoring. For each instantiation of these patterns templates (patterns over concrete delta events), one can generate a deterministic monitor that checks if the pattern is violated by some given trace. A pattern is said to be mined from a trace (or a collection of traces) if it is not violated by the trace(s). An approach for evaluating these instantiations efficiently is described in [20].

Now given a set of input-output traces and a pattern template, our technique first generates a pattern graph $G = (V, E \subseteq V \times V)$. A vertex $v \in V$ represents an event over $I \cup O$. For example, a vertex labeled with $\Delta a_{0\to 1}$ represents the event of signal a transitioning from 0 to 1. There is a directed edge $e = (u, v) \in E$ if and only if the pattern involving the delta events represented by u and v is satisfied in all traces.

As an example, let α be the arbiter described in Section 2.2.1. Suppose π uses a round-robin priority scheme and π' uses a fixed priority scheme for arbitration.

Given the input-output traces of π and π' and the **Until** pattern, suppose that Figure 2.10 shows the pattern graphs generated for π and π'.[9]

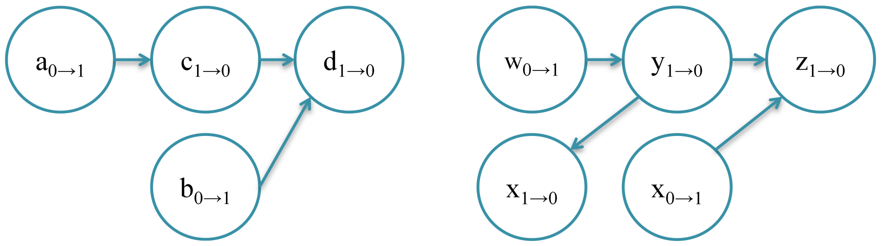


Figure 2.10: Arbiter Pattern Graphs

Table 2.1 shows how the named signals a-d of the round-robin priority arbiter and w-z of the fixed-priority arbiter correspond to request/grant signals r_0,r_1,g_0, and g_1 as described in Sec. 2.2.1. (Our task is to compute this correspondence, but we provide it here so that the reader can follow this example.) Thus, for example, the edges from $a_{0\to1}$ to $c_{0\to1}$ and from

[9]In the case where a pattern graphs is composed of multiple disconnected subgraphs, we use the biggest subgraph as the pattern graph.

$w_{0\to1}$ to $y_{0\to1}$ are instances of the property $\mathbf{G}\ [r_{00\to1} \Rightarrow \mathbf{X}\ (r_0\ \mathbf{U}\ g_{00\to1})]$ (request stays high until the corresponding grant is asserted).

Table 2.1: Signal Names in Arbiter Versions

Signals	Round-robin	Fixed
r_0	a	w
r_1	b	x
g_0	c	y
g_1	d	z

Graph Matching:

A graph $G' = (V', E')$ is an *induced subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' = E \cap (V' \times V')$. G is said to be *isomorphic* to G' if there exists a bijective function $f : V \to V'$ such that $\forall\,(u, v) \in E \iff (f(u), f(v)) \in E'$.

Given the two pattern graphs G and G' corresponding to π and π' respectively, a *common subgraph* is a graph which is *isomorphic* to induced subgraphs of G and G'. We wish to find a *maximum common subgraph* (MCS) between the two graphs, i.e. a common subgraph between G and G' that has the maximum number of vertices.

The key observation here is that the bijective function f that defines the MCS is exactly the signal correspondence mapping σ that we are looking for. In fact, when the vertices represent delta events, our approach can identify corresponding signals even if they are implemented with opposite polarities in the two circuits.

Consider our arbiter example. Figure 2.11 shows a maximum common subgraph of the pattern graphs given in Figure 2.10 illustrating all the request and response signals are mapped correctly using MCS approach.

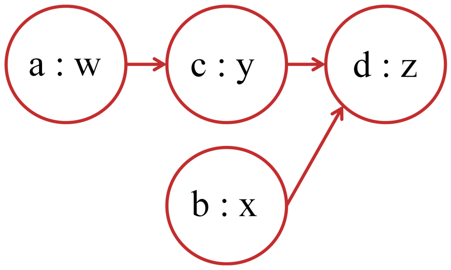


Figure 2.11: MCS in the two arbiter pattern graphs

The MCS problem is known to be NP-hard [15]. Most complete approaches are based on reformulating the problem into a *maximum clique* problem in a *compatibility graph* between G and G' [13]. The compatibility graph is a product graph of G and G' such that a vertex in the product graph is a pair of vertices (i, k) where $i \in V$ and $k \in V'$. There is an edge from (i, k) to (j, l) $(i \neq j$ and $k \neq l)$ if and only if one of the following conditions hold:

- $(i, j) \in E$ and $(k, l) \in E'$;
- $(i, j) \notin E$ and $(k, l) \notin E'$.

In a directed graph G, we say two vertices u and v are *connected* if both $(u, v) \in E$ and $(v, u) \in E$. A *clique* is then a subset of vertices such that every pair of vertices in this set are connected. A *maximum clique* is a clique with the most number of vertices.

We omit details of how the maximum clique problem is solved but refer the readers to [8]. The technique can scale to graphs with hundreds of vertices. Solving the maximum clique problem correctly generates the signal correspondence as depicted in Table 2.1.

Matching by Property Checking

If all inputs and outputs of the abstract component α appearing in its specification $\mathcal{S}$ are mapped onto some inputs and outputs of the unknown circuit $\mathfrak{n}$, then we proceed to the next step, where we verify whether $\mathfrak{n}$ satisfies $\mathcal{S}$. In our approach, we perform this step using model checking [11]. (If $\mathcal{S}$ were a reference circuit, it is possible to replace the model checker with a sequential equivalence checker.) If $\mathfrak{n}$ satisfies $\mathcal{S}$, then we terminate and report that $\mathfrak{n}$ matches the component α. Otherwise, we report that it does not match (and move to trying to match $\mathfrak{n}$ to a different abstract component).

However, if some inputs/outputs of α appearing in $\mathcal{S}$ are *not* matched to inputs/outputs of $\mathfrak{n}$, then we stop and declare that there is no match between $\mathfrak{n}$ and α. Note that this approach is *conservative*: it is possible for $\mathfrak{n}$ to be an instance of α but not pass this phase, since our input-output signal correspondence algorithm is heuristic. However, it is important to note that our matching approach is *sound* due to the use of formal verification.

While formal verification may not scale to the full circuit, we envision applying this procedure mainly to netlist slices with hundreds to thousands of library cells, which is within the capacity of state-of-the-art model checkers today. We demonstrate our approach on benchmarks from OpenCores, as discussed in Section 2.5.3.

Lastly, it should be noted that while it may be difficult to write specifications that account for all possible behaviors of each library component, it is generally possible to distinguish one design block from another using only a few logical specifications (e.g. distinguish an arbiter from an adder). We believe the automation that we provide in the proposed approach can benefit reverse-engineering greatly, as it is still mostly a manual process today.

2.4.3 Matching by Conditional Equivalence Checking

The previous section described an approach when a candidate module with proper boundaries is mapped to components in a predefined library. However, it is in general difficult to carve out such perfect modules. Additionally, we observe that due to optimizations performed during logic synthesis, netlist slices with different functions can overlap, i.e. some gates are shared between slices. For example, instead of having a separate set of gates implementing each opcode function, a single set of gates between the inputs and outputs of an ALU unit will suffice for all operations, each having overlapping logical block with another. In fact, gate-sharing can be a result of design choice and customization. The ripple-carry adder-subtractor design demonstrates this phenomenon where a single signal value can convert the adder to a subtractor. As a result, extra inputs are present when a netlist slice is carved out. Consequently, we can only reason about the function of the netlist slice when these *side inputs* are properly set. This means depending on the assignment to these side inputs, the netlist may or may not behave according to a certain operation, e.g. addition or subtraction. In this section, we describe an approach that addreses this problem, based on modifying the traditional equivalence checking problem into a quantified equivalence checking problem.

Generating Candidate Netlist Slices

Recall that we have all the words that can be inferred by propagation, we are now interested in finding computation structures that operate on these words. The main idea is to cut out the portion of the netlist that lies *between words*, and then check if this structure implements a particular word operation. We currently support only operations that are combinational logic. However, these still include many that are commonly found in circuit designs, such as addition, subtraction, Boolean operation (e.g. NOT, AND, OR, XOR) and shifting/rotation. Note that the user can extend this set with other word operations by providing reference models for those operations.

To extract the netlist between words, we first arrange the words in topological order between sequential boundaries. For example, given three words w^a, w^b, w^c of the same width such that w^a and w^b are followed by w^c in the topological order, and we are interested in checking if $w^c = w^a + w^b$ modulo a carry-in, we can form a netlist slice by using the gates between w^a and w^c and those between w^b and w^c.

QBF Formulation

Our solution is to model the problem as a Quantified Boolean Formula (QBF) and make use of state-of-the-art QBF solvers to solve it. This is different from the traditional SAT-based formulation for combintional equivalence checking [16], with the side inputs existentially quantified. QBF is the canonical complete problem for PSPACE. It extends propositional formulas by including the universal quantifier $\forall$ and existential quantifier $\exists$. The particular instance of QBF we have formulated here involves a single alternation of $\forall$ and $\exists$, which is also known as 2QBF. Figure 2.12 illustrates the miter construction for creating the 2QBF formula.

The reference circuit for the word operation we are interested in checking has inputs X. However, the extracted netlist slice also contains side inputs Y in addition to X. We can describe the miter circuit (as typically done for SAT-based combinational equivalence checking [16] in the verification literature) with a single Boolean formula ϕ such that ϕ is true if and only if the two circuits are equivalent. The intuition behind the existential quantification over Y is that if the netlist implements the function of the reference circuit,

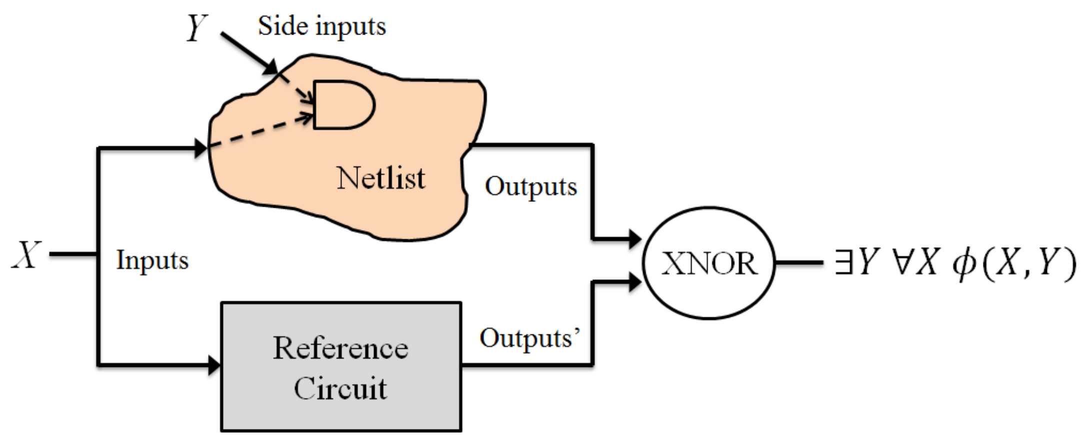


Figure 2.12: Miter Construction and QBF Formulation

then there must exist a way to configure the side inputs for it to do so. For the previous example of checking if $w^c = w^a + w^b$, we now have inputs as w^a and w^b and outputs as w^c. The formula ϕ desribes the comparison of the netlist with the disjunction of two circuits, one implements the addition with carry-in equal to 1, and the other with carry-in equal to 0.[10]

2.4.4 Discussion

We remark on the potential limitation of the function identification techniques and discuss open problems.

- Candidate module generation. We only partially address this problem in this paper – combinational blocks are carved out between words. However, the success of such matching by verification approach depends heavily on finding the correct input/output interfaces. In the case of matching by property checking, a partially matched module (when only a subset of the properties of the abstract component is satisfied) is still useful. In general, this is an open problem for us and we plan to explore it in future work.

- Robustness of pattern mining for solving I/O correspondence. We use pattern mining in this work. However, in general, any signature generation technique is applicable. In our experience, the simple pattern templates we use correspond to common hardware behaviorable patterns and thus are effective in capturing them.

- QBF formulation. Similar to word propagation, when the QBF procedure produces a satisfying assignment to the side inputs, the reachability of these values needs to

[10]We consider two possible reference circuits that the netlist can be matched to because the carry-in bit is not a primary input to the miter circuit.

Table 2.2: Benchmark Netitemize Information

Design	Gates	Nets	Latches	Description
router	896	984	182	CMP Router
open8	1807	1812	237	Open8 CPU
Cpu8080	2258	2368	243	8080 CPU
MIPS16	6986	11110	4380	MIPS-like core
oc8051	8093	10210	2748	8051 μcontroller
RISC FPU	14291	15740	3097	RISC FPU
BigSoC	375090	231736	34318	SoC

be checked. Again, due to scalability issues, we optimistically accept any satisfying assignment. Another limitation of the current QBF formulation is that it only applies to combinational netlist slices. For sequential circuits, we hypothesize that one can use a bounded enrolling to leverage the same QBF approach, and this is a subject of future work.

2.5 Experimental Results

We have developed an inference tool using Python and C++ that implements the algorithms described in this paper. The tool takes as input a synthesized netlist, analyzes it and produces as output word-level datapaths as well as a set of annotated netlist slices. The tool uses DepQBF [24] as the backend solver for solving QBFs.

2.5.1 Benchmarks

Table 2.2 summarizes the netlist information of the designs that are used in this paper. The generic name BigSoC is used for confidentiality reasons. All the designs were synthesized with an IBM/ARM cell library for a 45nm SOI process using the Synopsys Design Compiler with its default optimization setting.

The CMP router is a simplified version of the chip-multiprocessor router proposed in [28]. BigSoC is a system-on-a-chip design which consists of 7 subsystems: a 32-bit ARM-compatible RISC processor, a Singular Value Decomposition module, a SPI interface, a UART interface, an I^{2}C interface, a VGA controller and a memory controller. The subsystems are further interconnected through an AXI4S switch. The rest of the benchmarks are available on OpenCores [4].

We also separately evaluate the "matching-by-property-checking" approach and used the following three circuits obtained from OpenCores [4] in our experiments.

- WISHBONE-compatible SPI (WB-SPI) [7]
- WISHBONE-compatible SimpleSPI (WB-SimpleSPI) [5]

- WISHBONE-compatible I^2C (WB-I^2C) [2]

The serial peripheral interface (SPI) is a full duplex, serial communication link. Devices operate in either master or slave mode. Typically there is a single master device and one or more slave devices. SPI specifies four logic signals: SCLK (serial clock), MOSI (master output/slave input), MISO (master input, slave output), and SS (slave select). The SS signal is only necessary if more than one slave device is connected to the master. To initiate a data transfer, the SS pin for the desired slave is first pulled low. Then data are clocked from the master to the slave via the MOSI port and data are clocked from the slave to the master via the MISO port. When the transfer is complete, the SS pin is pulled high. SimpleSPI is a simplified version which supports only one master and one slave.

The inter-integrated circuit or I^2C is a serial two-wire communication bus. Devices operate in either master or slave mode, similar to SPI; however, there can be multiple masters on an I^2C bus. The bus is made up of two logic signals: SCL (clock) and SDA (data). A typical data transfer starts with a master sending a START bit along with a 7-bit address for the slave it wishes to communicate with, and a bit indicating a read or write operation. The slave responds with an ACK bit and proceeds to operate in either read or write mode, depending on the master's request. Once the transmission is over, the master sends a STOP bit.

WISHBONE is a communication interface for IP cores that enhances design reuse by enforcing compatibility between cores. Furthermore, WISHBONE is open source, which makes it easy for engineers to share hardware designs. As such, many projects on OpenCores, a website dedicated to open source hardware designs, include a WISHBONE interface. The protocol supports handshaking, single read/write cycles, block read/write cycles, and read-modify-write cycles. All three circuits are supposed to be WISHBONE compatible.

In this part of the experiment, we consider the scenario where the WISHBONE protocol [6] has been pre-characterized as a library component and the WB-SPI circuit is given as a concrete implementation of the WISHBONE protocol. We treat the WB-SimpleSPI and WB-I^2C as unknown candidate netlist slices. The goal is to determine whether an unknown circuit also implements the WISHBONE interface.

2.5.2 Finding Word-Level Flow

Table 2.3 summarizes the results of applying WordRev to the netlist designs. Columns 2 and 3 record the number of input and output words used respectively. Column 4 records the number of candidate words identified using the algorithms in Section 2.3.1. Column 5 is the number of words produced by word propagation. Column 6 is the runtime for word propagation in minutes. In all experiments, we limited the search to only words between 4 and 32 bits wide.

Input and output words were assumed to be known, since these are at the I/O of the design. Candidate words were selected from the word identification step described in Section 2.3.1. For BigSoC, we only selected 16 32-bit words identified by using the bitslice

Table 2.3: WordRev Statistics

Design	Input	Output	Cand.	Prop.	Runtime (min)
router	2	2	0	54	1.3
Open8	2	2	22	98	80.0
cpu8080	1	2	6	174	114.2
MIPS16	0	1	2	4	1.0
oc8051	6	7	12	113	329.9
RISC FPU	1	2	128	142	154.6
BigSoC	1	4	16	865	311.2

aggregation algorithm in Section 2.3.1. This is because a lot of words were identified as candidate words, which caused the tool to timeout as a result of trying to propagate each of them. Additionally, we consider words that can be propagated to be more valuable than just identified words, since propagation indicates that these arrays of bits are more likely to be operated together at the RTL level.

Case Study: CMP Router

In this section, we use the CMP router to evaluate the effectiveness of WordRev in detail. Particularly, we focus on the following analysis.

- Usefulness of the flow of word-level information presented as a directed graph.
- Effect of the netlist being synthesized from different cell libraries.
- Effect of the netlist being synthesized from the same cell library but with different optimization settings.

For convenience, we called the router netlist used in the previous section the “TR Router”, the resulting netlist synthesized from a different cell library the “TS Router”, and the optimized netlist the “Optimized TR Router”.

The overview of the CMP router design is illustrated in Figure 2.13. It is a composition of four high-level modules. The *input controller* comprises a set of FIFOs buffering incoming *flits* and interacting with the *arbiter*. A *flit* is a flow control unit. A data packet is composed of multiple flits. In this implementation, a flit has 12 bits, with the lower two bits as a header (used for channel reservation) and the remaining 10 bits as address (6 bits) and data (4 bits). Each *input controller* contains a circular FIFO buffer of 4 flits deep. When the arbiter grants access to a particular output port, it sends a signal to the input controller to release the flits from the buffers, and at the same time, it sends the allocation information to the *encoder* which in turn configures the *crossbar* to route the flits to the appropriate output port.

Word-Level Datapath: Figure 2.14 shows the word-level datapath as a directed graph produced by our tool. We have highlighted regions of the graph that correspond to high-level modules of the router as described previously. The nodes in yellow are the known words

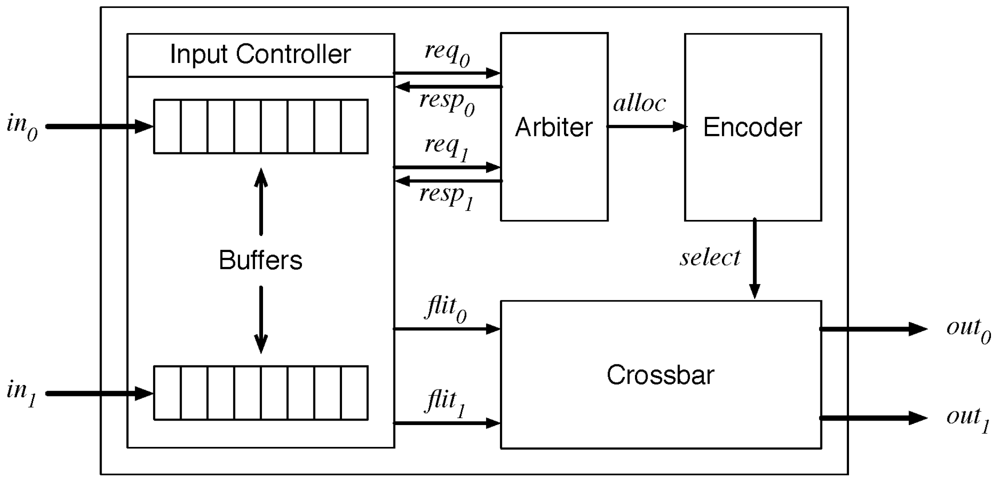


Figure 2.13: CMP Router Comprising Four High-Level Modules

that we start with at the primary input and output of the router. The number in each node denotes the numbering of the word (e.g. "w11") followed by the width of the word (e.g. 12 for "w11"). If a node is contained in another node, it indicates that the inner word is a subword of the outer word.

The top half of the word graph is in fact isomorphic to the bottom half, each corresponding to a port of the router. The input controller FIFO is the subgraph that has two special nodes, marked as "Writing into FIFO" and "Reading from FIFO". From the first node, which is the write pointer of the FIFO, it splits to four other words. Each of these words is then latched (as indicated by the red arrow). The latched output has three possible out-going paths: one going back to the itself (typical for state holding elements), one goes to the write pointer of the FIFO (multiplexed to determine whether it will get overwritten), and the last one goes to the read pointer of the FIFO (another multiplexing for determining which flit will be read). The crossbar on the right is easier to recognize – it consists of two 12-bit words that interweave to two other 12-bit words downstream. While the module identification described here was performed manually, comparing to Figure 2.13, the word graph in Figure 2.14 essentially reduces the router netlist with approximately 1100 cells to a single graph (which can fit on a page) that captures most aspects of the data-flow and architectural features of the router. This allows to a human analyst to look for patterns at a higher level of abstraction. Moreover, the word graph provides a structure for further automation, such as module identification, which we plan to explore in future work.

Different Cell Libraries: "TS Router" had about 4% less gates than "TR Router", but had the same number of latches.[11] Applying the same word propagation algorithm to "TS Router", 52 (instead of 54) words were found. Comparing the words inferred in the two

[11]The "TS" and "TR" cell libraries differ only in the layout implementation, where one is speed optimized and the other is size optimized. Their logical functions are the same.

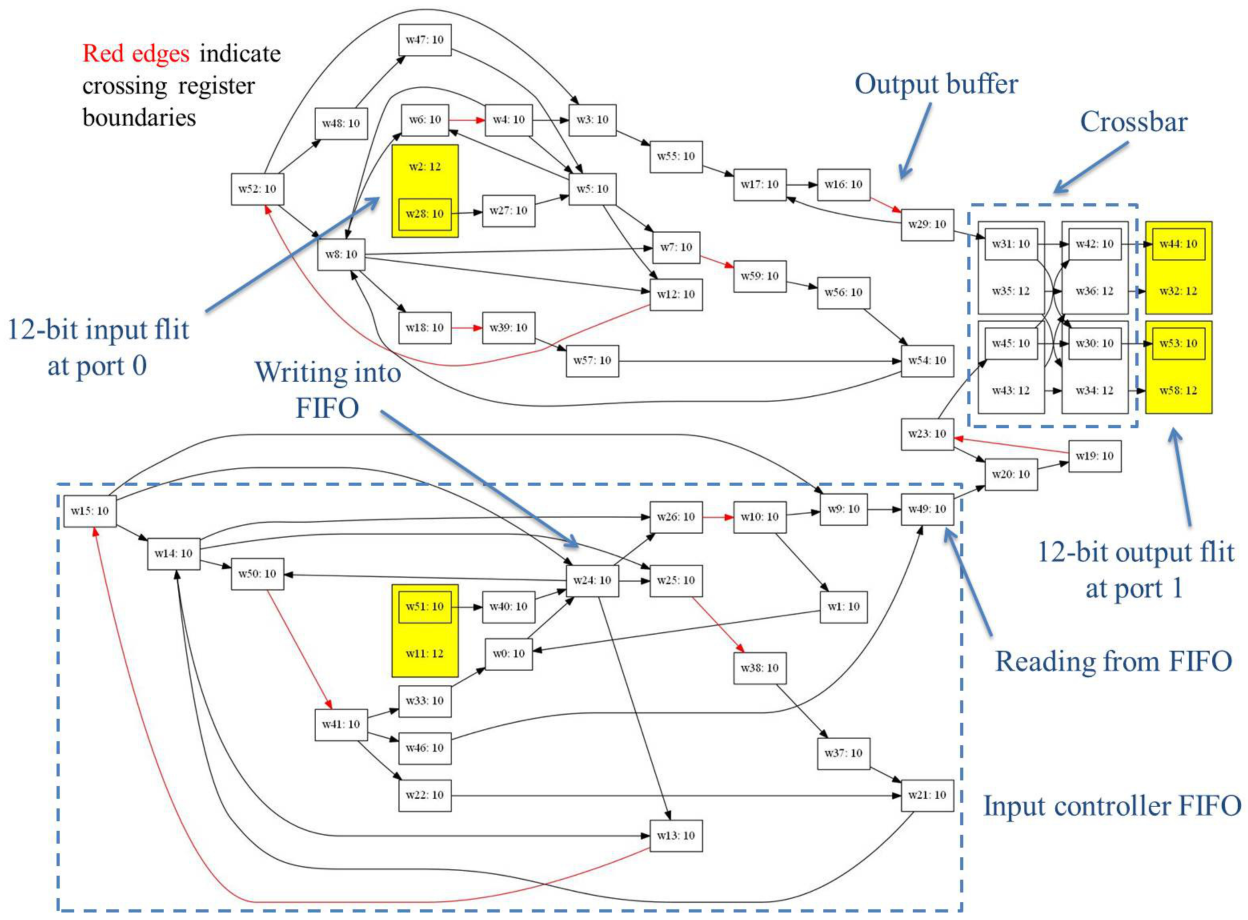

Red edges indicate crossing register boundaries
Output buffer
Crossbar
12-bit input flit at port 0
Writing into FIFO
12-bit output flit at port 1
Reading from FIFO
Input controller FIFO

Figure 2.14: Word-Level Datapath for the Same CMP Router Design

netlists directly was difficult since the wires were named differently. However, upon close examination of the word graphs, we verified that the topology of the "TS Router" word graph is almost identifical to that of "TR Router". The only differences were at the read and write pointers of the FIFOs. In addition, the widths of the words were also the same, showing the splitting of I/O words into words of 10 bits wide. This shows that our heuristic for finding target words to propagate is robust to small change in the cell library used in logic synthesis.

Different Synthesis Parameters: "Optimized TR Router" contained about 25% less gates than "TR Router".[12] In addition, it used 176 instead of 182 latches. In this case, WordRev found 50 words, 4 less than the number of words found in "TR Router". We analyzed the word graph generated for "Optimized TR Router" and found interesting discrepancies. While the topology of the graph remained largely the same, which means the key features were

[12] "Optimized TR Router" was the result of using the additional Synopsys "compile_ultra" command, which does extra optimizations for high-performance (i.e., high clock frequency) designs.

again visible, the words being propagated were only 4 bits wide. In fact, they correspond to the 4 data bits in each flit. In “TR Router”, both the 6-bit wide address field and the 4-bit wide data field were propagated together. This shows that while our algorithm can extract the word-level dataflow of a netlist, its performance can be influenced by the optimization setting during logic synthesis.

2.5.3 Functional Module Identification

QBF-Based Matching

In this section, we focus on reporting results that demonstrate the effectiveness of using the QBF formulation to identify structures and conditions in reasoning various word operations.

Table 2.4: QBF Statistics for Different Operations

Operation	Num of Vars	Num of Clauses	Runtime (s)
Addition	4525	11974	60
Subtraction	4438	11744	50
Rotate-right	4332	11416	30
Rotate-left	4332	11416	30
Not	4333	11415	64
And	4337	11428	34
Or	4337	11428	51

The OC8051 microcontroller is widely used in many embedded system products. It contains an 8-bit CPU optimized for control applications. Using word identification techniques described in Section 2.3.1 and the structure extraction process described in Section 2.4.3, we extracted a netlist slice that was part of the ALU unit of the microcontroller. This netlist slice contained 435 12SOI gates. In addition to the two 8-bit input words, the netlist slice had 87 other side inputs. We formulated a QBF problem for each of the word-level operation (each word is 8-bit wide and each operation corresponds to a specific ALU opcode) shown in Table 2.4, and verified that the *single* netlist slice could implement that operation by making appropriate assignments to the side inputs.

Property Checking with Unknown I/O Correspondence

Signal Correspondence:

As described in Section 2.4.2, we followed the approach in [20] for generating the pattern graph by mining the **Until** pattern on the simulation traces of each circuit. The simulation traces were produced by the test benches provided on OpenCores. We further assume the clock, reset (*wb_rst_i*) and data-path signals at the interface are already identified (these are easy to identify by structural methods). Next, a compatibility graph was generated for the pattern graph of the unknown circuit and WB-SPI.[13] The time taken to generate each pattern graph was under a second. We used the program `Cliquer` [27] which implements an exact branch-and-bound algorithm developed by Patric Östergård for finding the maximum clique in a graph. The time taken to find a maximum clique in the compatibility graph was under a second in all instances.

Table 2.5 shows the signal map derived between SPI and SimpleSPI by using the first MCS produced. All the WB-related signals were mapped correctly. In addition, two SPI-related signals were also mapped correctly. The signal names used here were taken from the respective RTL files and only serve as an illustration. In an actual reverse engineering exercise, the signal names of the unknown circuit will be arbitrary. These results were obtained despite the fact that the traces were generated by two test benches with very different behaviors.

Table 2.5: Signal Mapping between WB-SPI and WB-SimpleSPI

WB-SPI	WB-SimpleSPI
miso_pad_i	*miso_i*
wb_ack_o	*ack_o*
sclk_pad_o	*sck_o*
wb_we_i	*we_i*
wb_cyc_i	*cyc_i*
wb_stb_i	*stb_i*

Table 2.6 shows the signal map derived between SPI and I^{2}C by also using the first MCS produced.

Table 2.6: Signal Mapping between WB-SPI and WB-I^{2}C

WB-SPI	WB-I^{2}C
wb_we_i	*wb_we_i*
wb_cyc_i	*wb_cyc_i*
wb_ack_o	*wb_ack_o*

All the WB-related signal were again matched correctly. However, the *wb_stb_i* signal was not matched. It was identified by iterating the approach with the *Alternating* pattern, given the current matches. This suggests an incremental approach to our framework but this will be left to future work. On the other hand, there were also four other incorrectly matched signals. This is because other than being both WISHBONE compatible, the two circuits were actually implementing different functions. The next step which checks these signals against their logical specifications will eliminate the incorrect matches.

[13]Observe that the pattern graph only needs to be generated once for the library component.

Matching by Verification:

We focused on specifying the slave interface of the WISHBONE protocol, although we used properties of the master interface as assumptions.

At a minimum, a slave interface requires the following signals: wb_ack_o, wb_clk_i, wb_cyc_i, wb_stb_i, and wb_rst_i. Since the WISHBONE interface does not explicitly require the data lines, wb_dat_o and wb_dat_i, the only properties we could specify were about the reset operation and the handshaking protocol.

The properties for the reset operation and handshaking protocol are as follows:

- $\mathbf{G}\ (wb\_rst\_i \Rightarrow \mathbf{X}\ \neg wb\_ack\_o)$
- $\mathbf{G}\ \neg(wb\_ack\_o \wedge \mathbf{X}\ wb\_ack\_o)$
- $\mathbf{G}\ ((wb\_cyc\_i \wedge wb\_stb\_i) \Rightarrow \mathbf{F}\ wb\_ack\_o)$

The assumptions we made on the master interface, part of the specification, are as follows:

- $\mathbf{G}\ (wb\_rst\_i \Rightarrow (\neg wb\_stb\_i \wedge \neg wb\_cyc\_i))$
- $\mathbf{G}\ (wb\_stb\_i \Rightarrow wb\_cyc\_i)$

These LTL properties were manually translated from the WISHBONE documentation Rev. B.4 [6].

Taking the netlist descriptions of WB-SimpleSPI and WB-I^{2}C, we translated them to the SMV format and verified the properties described above using the Cadence SMV model checker [1]. For both circuits, all the properties passed. This confirmed that both indeed implemented the WISHBONE interface. The verification times were under 0.1 second for either benchmark.

Chapter 3

Conclusion

3.1 Summary

In this report, we have presented a portfolio of novel techniques for reverse engineering circuit netlists to their high-level descriptions. We are also the first to provide a formal definition for the netlist reverse engineering problem (REHLD), based on the notion of matching against abstract components. Many of the techniques proposed here have connections to techniques used in formal verification. For example, symbolic evaluation is used to infer word propagation, and property checking and equivalence checking are used to match a unknown circuit to a known abstract component. These techniques allow us to cope with challenges such as the netlist being unstructured and an intractably large implementation space exacerbated by scarce documentation.

While reverse engineering does not directly address the problems of trojan detection and isolation, it enables one to perform more targeted and comprehensive analysis once the high-level description of the netlist is obtained. We envision the techniques proposed in this report will be applicable in the grand picture of ensuring the integrity of a fabricated circuit, and will be a valuable complement to "dynamic" techniques that exist in the literature.

We have proposed an initial and integrated solution to the reverse engineering problem and the experimental results are promising. In fact, we tackled netlists that are orders of magnitude larger than those considered in previous reverse engineering approaches (e.g. [17]). However, we also recognize the potential limitations of the proposed techniques, as described in Section 2.3.3 and Section 2.4.4. We plan to address these problems in future work.

3.2 Acknowledgement

The author would like to thank his collaborators Prof. Sanjit A. Seshia, Zach Wasson and Wei Yang Tan from UC Berkeley, Adriá Gàscon, Dr. Natarajan Shankar and Dr. Ashish Tiwari from SRI International, and Pramod Subramanyan and Prof. Sharad Malik from

Princeton University. Some of the work done in this report could not have been possible without their assistance and guidance. In particular, the observation that setting a few "control" wires is likely to be sufficient for propagating the values of a word was first made by Dr. Ashish Tiwari. The work on word propagation also benefited much from discussion and working closely with Adriá Gàscon. The idea of using shapehashing is due to Dr. Natarajan Shankar. The author would like to thank Pramod Subramanyan and Prof. Sharad Malik from Princeton University for the help with their tool, BSIM, which was used in the word identification step. The author would also like to thank Prof. Sanjit A. Seshia and Prof. Robert K. Brayton for reviewing this report and giving insightful feedback. The work described in this report was supported in part by Defense Advanced Research Projects Agency (DARPA) under the IRIS program, and by the Hellman Family Faculty Fund.

Bibliography

[1] Cadence smv. http://www.kenmcmil.com/smv.html.

[2] I2c controller core. http://opencores.com/project,i2c.

[3] Integrity and reliability of integrated circuits (iris). http://www.darpa.mil/Our_Work/MTO/Programs/Integrity_and_Reliability_of_Integrated_Circuits_(IRIS).aspx.

[4] Opencores. http://opencores.org/.

[5] Simple spi core. http://opencores.com/project,simple_spi.

[6] Soc interconnection: Wishbone. http://opencores.org/opencores,wishbone.

[7] Spi core. http://opencores.com/project,spi.

[8] Egon Balas and Chang Sung Yu. Finding a maximum clique in an arbitrary graph. *SIAM J. Comput.*, 15:1054–1068, November 1986.

[9] R.E. Bryant. Symbolic simulation-techniques and applications. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 517 –521, Jun 1990.

[10] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam. Reducing structural bias in technology mapping. In *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, pages 519 – 526, Nov 2005.

[11] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2000.

[12] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton. Design patterns for reconfigurable computing. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 13 – 23, april 2004.

[13] P. J. Durand, R. Pasari, J. W. Baker, and C. Tsai. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 1999.

[14] Semiconductor Equipment and Materials Industry (SEMI). Ip challenges for the semiconductor equipment and materials industry. http://www.semi.org/sites/semi.org/files/docs/2012_IP_White_Paper.pdf. 2012.

[15] M. R. Garey and D. S. Johnson. *Computer and intractability*. Freeman, 1979.

[16] E. Goldberg, M. Prasad, and R. Brayton. Using sat for combinational equivalence checking. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '01, pages 114–121, Piscataway, NJ, USA, 2001. IEEE Press.

[17] Mark C. Hansen, Hakan Yalcin, and John P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.

[18] Matthew Hicks, Murph Finnicum, Samuel T. King, Milo M. K. Martin, and Jonathan M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 159 –172, May 2010.

[19] Y. Jin, N. Kupp, and Y. Makris. Experiences in hardware trojan design and implementation. In *Hardware-Oriented Security and Trust, 2009. HOST '09. IEEE International Workshop on*, pages 50 –57, july 2009.

[20] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. Scalable specification mining for verification and diagnosis. In *DAC '10*, pages 755–760, 2010.

[21] Wenchao Li, Adria Gascon, Pramod Subramanyan, Wei Yang Tan, Ashish Tiwari, Sharad Malik, Natarajan Shankar, and Sanjit A. Seshia. Wordrev: Finding word-level structures in a sea of bit-level gates. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, June 2013.

[22] Wenchao Li, Zach Wasson, and Sanjit A. Seshia. Reverse engineering circuits using behavioral pattern mining. In *Proceedings of the IEEE Conference on Hardware-Oriented Security and Trust (HOST)*, Jun 2012.

[23] Lang Lin, Markus Kasper, Tim Gneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 382–395. Springer, 2009.

[24] Florian Lonsing and Armin Biere. Depqbf: A dependency-aware qbf solver. *JSAT*, 7(2-3):71–76, 2010.

[25] J. Mohnke, P. Molitor, and S. Malik. Establishing latch correspondence for sequential circuits using distinguishing signatures. In *MidWest Symposium on Circuits and Systems*, pages 472–476, 1997.

[26] Janette Mohnke and Sharad Malik. Permutation and phase independent boolean comparison. In *European Conference on Design Automation*, Feb 1993.

[27] Sampo Niskanen and Patric Östergård. Cliquer - routines for clique searching. http://users.tkk.fi/pat/cliquer.html.

[28] Li-Shiuan Peh. *Flow control and micro-architectural mechanisms for extending the performance of interconnection networks*. PhD thesis, 2001.

[29] Amir Pnueli. The temporal logic of programs. pages 46–57, 1977.

[30] J. Paul Roth. *Computer Logic, Testing and Verification*. W. H. Freeman & Co., New York, NY, USA, 1980.

[31] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. EPIC: Ending piracy of integrated circuits. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1069–1074, 2008.

[32] Kristin Y. Rozier. Survey: Linear temporal logic symbolic model checking. *Comput. Sci. Rev.*, 5(2):163–203, May 2011.

[33] Yiqiong Shi, Chan Wai Ting, Bah-Hwee Gwee, and Ye Ren. A highly efficient method for extracting FSMs from flattened gate-level netlist. In *IEEE International Symposium on Circuits and Systems (ISCAS 2010)*, pages 2610–2613, 2010.

[34] C. Sturton, M. Hicks, D. Wagner, and S.T. King. Defeating uci: Building stealthy and malicious hardware. In *Security and Privacy (SP), IEEE Symposium on*, pages 64 –77, May 2011.

[35] Pramod Subramanyan, Nestan Tsiskaridze, Kanika Pasricha, Dillon Reisman, Adriana Susnea, and Sharad Malik. Reverse engineering digital circuits using functional analysis. In *Design Automation and Test in Europe (DATE)*, Mar 2013.

[36] M. Tehranipour and F. Koushanfar. A survey of trojan taxonomy and detection. *IEEE Design and Test of Computers*, 27:10 – 25, 2010 2010.

[37] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, pages 133–164. Elsevier, 1990.

[38] Randy Torrance and Dick James. The state-of-the-art in IC reverse engineering. In *11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 5747 of *Lecture Notes in Computer Science*, pages 363–381, 2009.